A Dope Fucking Cxx Cheat Sheet

Initialization

don't us this

`auto x = {n} | int y = {n}`

use this instead

`std::initializer_list<int> x = {n}`

some initilaization rules:

```
vector<int> x{10}      // one element - value = 10
vector<int> y(10, 20)  // 10 elements - value = 20
vector<int> z = 1      // Error. no conversion from int to vector


void f(const vector<double>&);

void g()
{
    v1 = 9; // error : no conversion from int to vector
    f(9);   // error : no conversion from int to vector
}

// By replacing " () " and " = "  w/ {} we get:
void g()
{
    v1 = {9}; // OK: v1 now has one element (with the value 9)
    f({9});   // OK: f is called with the list {9}
}
```

Limits

max Double

`double d = DBL_MAX`

max float

`float d = FLT_MAX`

these are defined in < climits >

max long double

`long double d = numeric_limits<long double>::max();`

max() is defined in < limits >

Conversion / Casting

Pointer, integral and floating-point values can be implicitly converted to bool
a nonzero value converts to true and zero value converts to false

```
/* e.g: */
void f (int* p, int i) {

    bool is_not_zero = p; // true if p!=0
    bool b2 = i; // true if i!=0
}
```

u can convert pointer to const pointer and reference to const reference implicitly

Casting Types

Static Cast

this is a compile time cast and does things like implicit conversion between types (int, float or pointer to void* and it can also call explicit conversion fuctions) (or implicit ones).

```
//e.g:
float a = 12.99;
int b = static_cast<int>(a);

// w/ *'s
int i = 12;
void *p = static_cast<void *>(&i);
int *x = static_cast<int *>(p);
```

Const Cast

const cast is used to cast away the constness of variable, const cast can be used to change non-const class members inside a function, const cast can be used to pass const data to a function that doesn't allow const

```
//e.g:

int f (int* p) {
    return (*ptr + 10);
}

const int val = 10;
const int *p = &val;

// u cannot modify the calue here
int *pf = const_cast<int *>(p);

std::cout << f(pf);

// if u wanna modify the value make int val non const:
int val = 10;

// same as above...
```

Dynamic Cast

dynamic cast works at runtime rather than compile time like static cast, DC can work only in polymorphic types

```
strutc A {
    virtual void f();
};
struct B : public A {};
struuct C {};

A a;
B b;

A *ap = &b;
B *b1 = dynamic_cast<B *>(&a);        // NULL, bcus 'a' is not 'b'
B *b2 = dynamic_cast<B *>(ap);        // 'b'
C *c  = dynamic_cast<C *>(ap);        // NULL

A& ar = dynamic_cast<A&>(*ap);        // OK.
B& br = dynamic_cast<B&>(*ap);        // OK.
C& cr = dynamic_cast<C&>(*ap);        // ERROR: std::bad_cast
```

A dynamic_cast requires a pointer or a reference to a polymorphic type in order to do a downcast or a crosscast.

`example`

```
class My_slider: public Ival_slider { // polymor phic base (Ival_slider has virtual functions)
    // ...
};

class My_date : public Date { // base not polymorphic (Date has no virtual functions)
    // ...
};

void g(Ival_box* pb, Date* pd) {
    My_slider* pd1 = dynamic_cast<My_slider*>(pb); // OK
    My_date* pd2 = dynamic_cast<My_date*>(pd); // error : Date not polymorphic
}
```

Requiring the pointer's type to be polymorphic simplifies the implementation of dynamic_cast because it makes it easy to find a place to hold the necessary information about the object's type. A typical implementation will attach a “type information object” to an object by placing a pointer to the type information in the virtual function table for the object's class.

`example`

```
My_Slider
     _______
    |       |<----
    |  ...  |     \  Vtable          Type Info
    |  Vptr |      \ ______          _______________
    |  ...  |_____\ |      |______\ |               |               Type Info
    |_______|     / |      |      / |  "My SLider"  | _____________
                    |      |        |    bases      |            \/ _______________
                    |      |        |_______________|           |  "laval_slider"  |
                    |      |_________                           |__________________|
                    |______|        \/

                                     ________________________
                                    | My_Slider::get_value() |
                                    |________________________|
```

A dynamic_cast to **void*** can be used to determine the address of the beginning of an object of polymorphic type.

`exmaple`

```cpp
void g(Ival_box* pb, Date* pd)
{
    void* pb2 = dynamic_cast<void*>(pb); // OK
    void* pd2 = dynamic_cast<void*>(pd); // error : Date not polymorphic
}
```

Note: **There is no dynamic_cast from void*** because there would be no way of knowing where to find the vptr

Reinterpret Cast

RC is used to convert one pointer of another pointer of any type, no matter wether the class is related to each other or not. it does not check if the pointer type and the data pointed to by the pointer is same or not. and it doesn't have any return type it simply converts the pointer.

syntax
`datatype *var_name = reinterpret_cast\<data_type *>(pointer variable);`

```cpp
int *p = new int(64);
char *cp = reinterpret_cast<char *>(p);

cout << *p << '\n';      // 64;
cout << *cp << '\n';     // B;
cout <<  p << '\n';      // 0x902bf;
cout <<  ch << '\n';     // A;
```

Convert string to other data types

ima just put some basic use cases and shit here.

```cpp
auto a = std::stoi("12");    // string -> int
auto b = std::stof("12");    // string -> float
auto c = std::stod("12");    // string -> double
auto d = std::stol("12");    // string -> long
auto e = std::stoll("12");   // string -> long long
auto f = std::stoul("12");   // string -> unsigned long
auto g = std::stoull("12");  // string -> unsigned long long
auto h = std::stoll("12");   // string -> long long
```

also it discards any characters in a string.

`example`

```cpp
std::string str = "Free Tekashi 69";
auto val = std::stoi(str);

// val: 69
```

theres still a lot of things like bases and position and shit so, just refer to [cppreference]() if u want greater details bout it.

Errors

throw runtime error

```cpp
if (1 != 2)
    throw std::runtime_error("What am i crazy");
```

Operators

USE	**SYNTAX**
Subscripting	pointer -> member
Member selection	pointer [expr]
Function call	expr (expr-list)
Value construction	type { expr-list }
Function-style type conversion	type (expr-list)
Post increment	lvalue ++
Post decrement	lvalue --
Type identification	typeid (type)
Run-time type identification	typeid (expr)
Run-time checked conversion	dynamic_cast < type > (expr)
Member selection	object . member
Compile-time checked conversion	static_cast < type > (expr)
Unchecked conversion	reinterpret_cast < type > (expr)
const conversion	const_cast < type > (expr)
Size of object	sizeof expr
Size of type	sizeof (type)
Size of parameter pack	sizeof... name
Alignment of type	alignof (type)
Pre increment	++ lvalue
Pre decrement	-- lvalue
Complement	˜ expr
Not	! expr

Unary minus	− expr
Unary plus	+ expr
Address of	& lvalue
Dereference	∗ expr
Create (allocate)	new type
Create (allocate and initialize)	new type (expr-list)
Create (allocate and initialize)	new type { expr-list }
Create (place)	new (expr-list) type
Create (place and initialize)	new (expr-list) type (expr-list)
Create (place and initialize)	new (expr-list) type { expr-list }
Destroy (deallocate)	delete pointer
Destroy array	delete [] pointer
Can expression throw?	noexcept (expr)
Cast (type conversion)	(type) expr
Member selection	object .∗ pointer-to-member
Member selection	pointer −>∗ pointer-to-member
Multiply	expr ∗ expr
Divide	expr / expr
Modulo (remainder)	expr % expr
Add (plus)	expr + expr
Subtract (minus)	expr − expr
Shift left	expr << expr
Shift right	expr >> expr
Less than	expr < expr
Less than or equal	expr <= expr
Greater than	expr > expr
Greater than or equal	expr >= expr
Equal	expr == expr
Not equal	expr != expr
Bitwise and	expr & expr
Bitwise exclusive-or	expr ˆ expr
Bitwise inclusive-or	expr \ expr
Logical and	expr && expr
Logical inclusive or	expr \ \ expr
Conditional expression	expr ? expr : expr
List	{ expr-list }
Throw exception	throw expr
Simple assignment	lvalue = expr
Multiply and assign	lvalue ∗= expr
Divide and assign	lvalue /= expr
Modulo and assign	lvalue %= expr
Add and assign	lvalue += expr
Subtract and assign	lvalue −= expr
Shift left and assign	lvalue <<= expr
Shift right and assign	lvalue >>= expr
Bitwise and and assign	lvalue &= expr
Bitwise inclusive-or and assign	lvalue \ = expr
Bitwise exclusive-or and assign	lvalue ˆ= expr
comma (sequencing)	expr , expr

Token Summery

Token Class	**Example**
Character literal	vector , foo_bar , x3
Integer literal	int , for , virtual
Identifier	'x' , \n' , 'U'\UFADEFADE'
Floating-point literal	12 , 012 , 0x12
Keyword	1.2 , 1.2e−3 , 1.2L
String literal	"Hello!" , R"("World"!)"
Operator	+= , % , <<
Punctuation	; , , , { , } , (,)
Preprocessor notation	# , ##

Alternative Representation

TYPEDEF	**OPERATOR**
and	&
and_eq	&=
bitand	&
bitor	\
compl	~
not	!
not_eq	!=
or	\ \
or_eq	\ =

xor ^
xor_eq ^=

Shortcut to copy strings

```
while (*p++ = *q++);
```

Function Pointer

function pointer is a pointer that points to a function instead like how a normal pointer points to a variable or an object. obviously duh!
please reffer to [learncpp] for greater details. cus i'm lazy af and i aint typin all of it.

- when a function is called the execution jumps to the address of that function. cus like variables functions live at assigned address in memory.

`example`

```
int foo() {      // suppose code for foo starts at memory address 0x7981937
    return 5;
}

main() {
    foo()   // execuion jumps to 0x7981937
}
```

- syntax for creaing non-const function pointer, whose return type is int. and takes no arguement `int (*anyName)()` this f pointer can point to any funcion that matches the type.
- syntax for defining a funcion poiner that takes a const int `int (*const int)()` note that u have to put const after the abstreck and if u put cons before the int like `const int (*int)()` this would mean that this function returns a const int. again "duh!"

`assigning function pointers`

```
int foo() { return 5; }
int goo() { return 5; }

main() {

    int (*fPtr)() = foo;     // fPtr points to foo()
    fPtr = goo;              // now fPtr points to goo()

    // plz dont do this -> fPtr = goo();
}
```

`calling function using funcion pointer`

- using explicit dereference

```
int foo(int x) { return x; }

main() {
    int (*fPtr)(int) = foo; // assing fPtr to foo()
    (*fPtr)(12);            // call foo() via fPtr
}
```

- using implicit dereference

```
int foo(int x) { return x; }

main() {
    int (*fPtr)(int) = foo; // assing fPtr to foo()
    fPtr(12);               // call foo() via fPtr
}
```

- you cant use default values with funcions pointers cus they are evaluated at run time and normal functions are evaluated at compile time.

```
int foo(int x = 2) {
    return x;
}

int main() {

    int (*fPtr)(int);

    fPtr = foo;      // assing fPtr to foo()
    fPtr();          // Error. cant use default value.

    foo();           // Ok. can use default value

}
```

- using function pointers as a callback function. so in this example we gon make a program which sorts an array of ints. and depending on the callback funcion that the user is gon provide we it gon order it (ascending / descending).

```cpp
#include <algorithm>
#include <iostream>

/*
    aphcourse we can make cmpFnctn take an default function
    for example bool (*cmpFnctn)(int, int) = asc
    if user do not provide the cmp function then its gon use default.
*/

void selecnSort(int *arr, int size, bool (*cmpFnctn)(int, int))
{
    for (int i = 0; i < size; ++i) {

        // bIndex if the smallest / largest elem we've encountered so far.
        int bIndex = i;

        for (int j = i + 1; j < size; ++j) {

            // check idf he new elem is smaller / larger than the previous elem.
            if (cmpFnctn(arr[bIndex], arr[j]))
                // this is new smallest / largest no. for this iteration
                bIndex = j;
        }

        std::swap(arr[i], arr[bIndex]);
    }
}

// comparison function that sorts in ascending.
bool asc(int x, int y) {
    return x > y;
}


// comparison function that sorts in descending.
bool dsc(int x, int y) {
    return x < y;
}

int main() {

    int arr[9] = {2, 3, 54, 12,  1, 99, 21, 6, 8};

    selecnSort(arr, 9, asc);
    std::cout << "Asc: \n";
    for (auto& x : arr)
        std::cout << x << '\n';

    selecnSort(arr, 9, dsc);
    std::cout << "Desc: \n";
    for (auto& x : arr)
        std::cout << x << '\n';

    return 0;
}
```

using `typedefs` and `using` statements to make em look cool

```cpp
typedef bool (*fPtr)(int);
using fPtr1 = bool (*)(int);

void f(fPtr fp);
void f1(fPtr1 fp);
```

Using `std::function` (introduced in Cxx 11)

std::function is an alternative for stoaring function pointers in Cxx, which is defined in std lib <functional>

`syntax`

`std::function<return type(arguement types if any else empty)>`

`example`

```cpp
#include <functional>

int foo(int x) {
    return x;
}

int main() {

    std::function<int(int)> fPtr = nullptr;
    fPtr = foo;

    fPtr(12);
}
```

Using `auto`

however using auto is error prone, cus return type and the arguments of the function or not exposed like they do w/ ***typedef***, ***using***, ***std::function***. tho its cool if u dont give a fug bout it and also if ur lazy af.

`example`

```
int foo(int x) {
    return x;
}

int main() {

    auto fPtr = foo;

    foo(12);
}
```

OS MACROS C/C++

```
_WIN32          // Windows 32 Bit
_WIn64          // Windows 64 Bit
__unix          // Unix
__unix__        // Unix
__APPLE__       // Apple
__MACH__        // Mac OS
__LINUX__       // Linux
__FreeBSD__     // Free BSD
```

`example`

```
std::string getOs() {

    #ifdef _WIN32
        return "Windows 32-bit";

    #elif _WIN64
        return "Windows 64-bit;

    #elif _unix || __unix__
        return "Unix";

    #elif __APPLE__ || __MACH__
        return "Mac OSx";

    #elif __LINUX__
        return "Linux";

    #elif __FreeBSD__
        return "FreeBSD";

    #else
        return "Other";

    #endif

}
```

Types of function declerations

- **inline** - indicating a desire to have function calls implemented by inlining the function body
- **constexpr** - indicating that it should be possible to evaluate the function at compile time if given constant expressions s arguments
- **noexcept** - indicating that the function may not throw an exception
- **static** - used for linkage specification
- **[[noreturn]]** - indicating that the function will not return using the normal call/return mechanism
- **virtual** - indicating that it can be overridden in a derived class
- **override** - indicating that it must be overriding a virtual function from a base class
- **final** - indicating that it cannot be overriden in a derived class
- **static** - indicating that it is not associated with a particular object
- **const** - indicating that it may not modify its object

```
// Little snippet of what you are getting youself into.
struct S {
    [[noreturn]] virtual inline auto f(const unsigned long int *const) -> void const noexcept;
};
```

You can also define return type of a function like following

```
auto f(void) -> int { } // int is a return type of this function.
```

Rule of thumb for passing values

- Use pass-by-value for small objects.
- Use pass-by- const -reference to pass large values that you don't need to modify.
- Return a result as a return value rather than modifying an object through an argument.
- Use rvalue references to implement move and forwarding.
- Pass a pointer if "no object" is a valid alternative (and represent "no object" by nullptr).
- Use pass-by-reference only if you have to.

Specify the size of the array while passed it to a fucntion

```
void f(int(&arr)[4]); // size = 4

int arr[] = {1, 2, 3};
f(arr); // OK.

int arr2[] = {1, 2, 3, 4};
f(arr); // Error. size !4


/*
    * Example w/ template
    *
    */

template <class T, int N>

void f(T(&errc)[N]) {

    for (auto& x : errc)
        std::cout << x << '\n';
}


int arr[] = {1, 2, 3, 4};
f<int, 4>(arr); // OK.

f(int, 5)(arr); // Error. size != size of the arr
```

Ellipsis

syntax:

```
return_type function_name (arg_list, ...)
```

- The arg_list is one or more normal function parameter
- Function that have ellipsis must have at least one non-ellipsis parameter
- Any parameter passed to the function must match the arg_list parameter first
- (...) must always be the last parameter in the function
- ellipsis are defined in <cstdrag> header

```
Exmaple
```

```cpp
#include <iostream>
#include <cstdarg>

// argc is how many additional args we are passing

double findAvg(int argc, ...) {

    double sum = 0;

    // we access the ellipsis thru va_list

    va_list ls;

    // we, initialize the va_list using va_start so,
    // first parameter is : the list to initialize
    // second parameter is : the last non - ellipsis

    va_start(ls, argc);

    // Loop thru all the ellipsis arguements
    for (int i = 0; i < argc; ++i)
        // we use va_arg to get parameters out of our ellipsis
        // First parameter is the va_list we are using
        // Second parameter is the type of the parameter

        sum += va_arg(ls, int);

    va_end(ls);

    return sum / argc;

}


int main() {

    std::cout << findAvg(5, 1, 2, 3, 4, 5) << '\n';
    std::cout << findAvg(6, 1, 2, 3, 4, 5, 6) << '\n';

    return 0;
}
```

```
Output:
3
3.5
```

but this is the shittiest thing you could do u cus type checking aint a thing in this hoe. so, i think we shoul use decoder convention to check for the type

Example

```cpp
#include <iostream>
#include <cstdarg>
#include <string>

// argc is how many additional args we are passing
double findAvg(std::string decoder, ...) {

    double sum = 0;

    // we access the ellipsis thru va_list
    va_list ls;

    // we, initialize the va_list using va_start so,
    // first parameter is : the list to initialize
    // second parameter is : the last non - ellipsis
    va_start(ls, decoder);

    int cnt = 0;

    while (1) {

        char codeType = decoder[cnt];

        switch (codeType)
        {
            default:

            case '\0':
                // clean up the va_list when we are done
                va_end(ls);
                return sum / cnt;
                break;

            case 'i':
                sum += va_arg(ls, int);
                ++cnt;
                break;

            case 'd':
                sum += va_arg(ls, double);
                ++cnt;
                break;
        }

    }

}


int main() {

    std::cout << findAvg("iii", 5, 3, 2) << '\n';
    std::cout << findAvg("iid", 1, 1, 2.3) << '\n';

    return 0;
}
```

```
Output:
3.33333
1.43333
```

Some predifined Macros

- `__cplusplus` defined in a C++ cpmpilation (not in C), its value is 201103L
- `__DATE__` date in ***yyyy:mm:dd*** format
- `__TIME__` time in ***hh:mm:ss*** format
- `__FILE__` name of the current source file
- `__FUNC__` an implementation defined C style string, naming the current function
- `__STDC_HOSTED` ***1*** if the implementation hostd, otherwise ***0***
- `__STDC__` defined in C compilation (not in C++)
- `__STDC_MB_MIGHT_NEQ_WC__` ***1*** if, in the encoding for ***wchar_t***, a member of the basic character set might have a code value differs from its value as an ordinary character litteral
- `__STDCPP_STRICT_POINTER_SAFETY__` ***1*** if the implementation has strict pointer safety, otherwise ***undefined***
- `__STDCPP_THREADS` ***1*** if the program can have more than one thread of execution, ohterwise ***undefined***

Memory measurement chart

Data Measurement	**Size**
Bit	single bit (1 or 0)
1 - Byte	8 Bits
1 - Kilobyte	1024 Bytes
1 - Megabyte	1024 Kbs
1 - Gigabyte	1024 Mbs

1 - Terabyte	1024 Gbs
1 - Petabyte	1024 Tbs
1 - Exhabyte	1024 Pbs

Standard Cxx Shit

Check if the type is polymorphic w/ std::is_polymorphic

i.e if class or struct has atleast one virtual function or derived from it

```
#include <type_traits>
#include <iostream>

struct Foo { };

struct Bar {

    virtual void bar();
};

struct Baz : Bar { };

int main() {

    std::cout << std::boolalpha
              << std::is_polymorphic<Foo>::value    // false
              << std::is_polymorphic<Bar>::value    // true
              << std::is_polymorphic<Baz>::value    // true
              << std::endl;

        // is_polymorphics<T>::value has the return value its just some metaprogramming shit u might wanna peek in that hoe.
}
```

compile time type eval w/ std::conditional

dont need to trip on this thick bitch its just a compile time selecto between two alternatives. if its first arg evaluated to true the result (presented as the member type) is the second argument; otherwise, the result is the third argument.

```
#include <type_traits>
#include <iostream>
#include <typeinfo>

int main() {

    using Type = std::conditional<false, int, double>::type;
    using Type2 = std::conditional<false, int, double>::type;

    std::cout << typeid(Type).name() << '\n';     // int
    std::cout << typeid(Type2).name() << '\n';    // double

    // all this conditional is doing is figuring out should the Type be int or double based on a condition that we gave it as the
first parameter

  return 0;
}
```

example above looks kinda dumb but look down here this could be fucking usefull eh?

```
struct Scoped { /* store shit on stack */ };
struct Heap { /* store shit on Free Store */ };

template<typename T>
struct Container {

  using type = typename std::conditional<(sizeof(T)<=on_stack_max),
                      Scoped<T>,   // first alternative
                      Heap<T>   // second alternative
                      >::type;

  static const int on_stack_max = /* some number */

  private:
    T* t;
};
```

above example is aphcouse aint gon compile but it illustrates how we can use condisional to our advantage. enought small talk the example above is kinda like std::string if the value if less thna on_stack max then store it on stack else store it on heap.

---

check is the type is same w/ std::is_same

this shit is fucking straight forward thank fucking keanu reeves the name aint smn crazy

```
#include <type_traits>
#include <iostream>

int main() {

  using IINNTT = char;
  using type = std::conditional<true, int, double>::type;

  std::cout << std::boolalpha
            << std::is_same<int, IINNTT>::value << '\n'
            << std::is_same<int, type>::value << '\n';

    // ps:  u cant use this shit with values u can only use it w/ types

  return 0;
}
```

Stop Execution of the program for some time

```
#include <iostream>
#include <chrono>
#include <thread>


void Matrix() {

    while (1) {

        std::cout << "Doing...\n";
        std::this_thread::sleep_for(std::chrono::seconds(3));
        std::cout << "After 3\n";
    }
}
```

Color and text Preferences

`Unix Specific`

`"\033[1;31m" <cutom text> "\033[0m"`

Color	FG	BG
black	30	40
red	31	41
green	32	42
yellow	33	43
blue	34	44
magenta	35	45
cyan	36	46
white	37	47

USE	VAL
reset	0 (everything back to how it was)
bold/bright	1
underline	4
Inverse	7 (swap foreground and background color)
Bold/bright off	21
underline/line off	24
inverse off	27

Get size of console window

`Windows`

```
#include <windows.h>

main() {

    CONSOLE_SCREEN_BUFFER_INFO csbi;
    GetConsoleScreenBufferInfo(GetSstHandler(STD_OUTPUT_HANDLE), &csbi);

    int cols = csbi.srWindow.Right - csbi.srWindow.Left + 1;    // Width
    int rows = csbi.srWindow.Bottom - csbi.srWindow.Top + 1;    // Height
}
```

`Linux`

```
#include <sys/ioctl.h>
#include <unistd.h>

main() {

    struct winsize sz;
    ioctl(STDOUT_FILENO, TIOCGWINSZ, &sz);

    sz.ws_row   // Height
    sz.ws_col   // Width
}
```

Assertions

Assertions are statements used to test assumptions made by programmer, for example we may use assertion to check wether pointer returned by malloc is NULL or not. If the expression evaluates to 0 (false) then expression, sourcode filename, line number are sent thru the stderr and then abort function is called

`Syntax`

`void assert( int expression );`

Example

```
#include <iostream>
#include <cassert>

void main() {

    int x = 7;

    /* some big code here...
       and lets say x accidently changed to 9
    */

    x = 9;

    // programmer assumes x to be 7 in rest of the code

    assert(x == 7);

}
```

```
// output
Assertion failed: x == 7, test.cpp, line 13
```

we can completely ignore assertion statements using **NDEBUG** macro

```
#define NDEBUG
#include <cassert>


void main() {

    int x = 7;

    x = 9;

    assert(x == 7);

}

// above code compiles and runs fine!
```

`static_assert`

`Syntax`

```
static_assert( bool_constexpr , message )      // since c++11
static_assert( bool_constexpr )      // since c++17
```

this* unconditionally checks its condition at comoile time, if the assertion fails compiler writes out the message and compilation fails

`example`

```cpp
#include <type_traits>

template <class T>

void swap(T& a, T& b) {

    // these is... are defined in < type_traits >

    static_assert(std::is_copy_constructable<T>::value, "swap requires copying");
    static_assert(std::is_nothrow_copy_constructable<T>::value && std::is_nothrow_copy_assignable<T>::value, "swap requires nothrow 
copy/assign");

    auto c = b;
            b = a;
            a = b;
}

struct nc {

    nc (const nc& ) = delete;
    nc() = default;
}

main() {

    int a, b;
    swap(a, b)      // Ok.

    struct nc nc_a, nc_b;
    swap(nc_a, nc_b);       // assertion faile: message: swap requires copying.
}
```

Convert boolean expressions into strings

by default 1 : true and 0 : false so, if wanna get it in a string litteral we can use std::boolalpha

```cpp
int b = true;
std::cout << b;                         // 1
std::cout << std::boolalpha << x       // true
```

Invoke Functions After Returnting from the main()

atexit(void*());

```cpp
// is invoked when exit() is called or returned from main

// e.g:

void handler() { std::cout << "Program Terminated\n"; }

int main() {

    // register the handler
    std::atexit(&handler);

    std::cout << "Returning From main() \n";
    return 0;
}
```

```
//  output
Returning From main()
Program Terminated
```

at_quick_exit(void(*));

```cpp
// this bastard is used to fuck w/ quick_exists
// e.g:

void handler() { std::cout << "Fuck Yeah Babie!\n"; }

int main() {

    at_quick_exit(&handler);

    std::cout << "leaving toilet w/t cleaning the shit.\n";

    std::quick_exit(0); // leave toilet w/t cleaning the shit.
}
```

```
// output
leaving toilet w/t cleaning the shit.
Fuck Yeah Babie!
```

---

explicit

explicit initialization is also known as direct initialization, A constructor declared with the keyword explicit can only be used for initialization and explicit conversions.

example

```
class Date {
        int d, m, y;
    public:
        explicit Date(int dd =0, int mm =0, int yy =0);
    // ...
};

Date d1 {15};            // OK: considered explicit
Date d2 = Date{15};      // OK: explicit
Date d3 = {15};          // error : = initialization does not do implicit conversions
Date d4 = 15;            // error : = initialization does not do implicit conversions
```

---

const member functions

The const after the (empty) argument list in the function declarations indicates that these functions do not modify the state of a Class .
in simle words if we got a private memeber for ex. x in const function we cannot modify it for ex. do ++x or x += 1, one thing to remeber bout const functions is that they are read only objects.

Example

```
class Date {
    int d, m, y;
    public:
        int day() const { return d; }
        int month() const { }
        int year() const;

        void add_year(int n);   // add n years
        // ...
};

int Date::year() const
{
    return ++y; // error : attempt to change member value in const function
}

int Date::month const {
    return m + 1    // Ok. just returning m + 1 not messing w/ m's value
}
```

imp thing bout the const is that if u make a object const then its functions and shit also has to const member functions ig.

```
// e.g:

class Baw {

    public:
        void smn() { /* ... */}

        void scnd() const { /* ... */ }
}

const Baw B;

B.smn();     // Error: smn is not a const member function
B.scnd();    // Ok.
```

---

mutable

mutable is a storage class specifier just like for ex: static, register, extern and auto. Sometimes there is requirement to modify one or more data members of class / struct through const function even though you don't want the function to update other members of class / struct. This task can be easily performed by using mutable keyword.

example

```
class Date {
    int d, m;
    mutable int y;

    public:
        int day() const { return d; }
        int month() const { }
        int year() const;

        void add_year(int n);   // add n years
        // ...
};

int Date::year() const {

    return ++y; // Ok. we good here, cus its mutable we can fuck w/ it.
}

int Date::month const {

    return m += 1    // error : attempt to change member value in const function
}
```

---

static member

A variable that is part of a class, yet is not part of an object of that class, is called a static member. There is exactly one copy of a static member instead of one copy per object, as for ordinary non- static members Similarly, a function that needs access to members of a
class, yet doesn't need to be invoked for a particular object, is called a static member function.

`example`

```
#include <iostream>

class F {
    int x;
    static F f;

    public:
        F(int );

        void doit() {
            std::cout <<  x;
        }

        static void smn() {
            std::cout << "i'm static\n";
        }

        static void setDefault(int n) {
            f = {n};    // set default value for F
        }
};

F F::f{20};     // set the value of f which would be out default value


F::F(int s) {
    x = s ? s : 1;
};

int main() {

    F x({});

    x.doit();   // Ok.

    x.smn();    // Ok.

    F::smn();   // OK.

    F::doit();  // Error: cannot call member function ‘void F::doit()’ without object


    return 0;
}
```

```
// output: ( aphcourse if we dont do F::doit() )
20i'm static
i'm static
```

---

print UTF-8 characters to the console

```
std::locale old_locale;  // current locale
setlocale(LC_ALL, "en_US.UTF-8");
wchar_t w = 0x262F;
std::wcout << w << std::endl;
setlocale(LC_ALL, old_locale.name().c_str());
```

Delegating Constructor

a member-style initializer using the class's own name (its constructor name) calls another constructor as part of the construction. Such a constructor is called a delegating constructor (and occasionally a forwarding constructor).

```
// Ex:
class X {
    int a;
public:
    X(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
    X() :X{42} { }
    X(string s) :X{to<int>(s)} { }
};
```

`= delete` and `= default`

default states that we are allowing the default copy, move or destaructor (which compiler provides us by defualt). **If a class has a pointer member, it probably needs a destructor and non-default copy operations**

Rules: 5 Statements Statements

`example`

```
class Foo {
public:
    Foo();                     // default constructor
    Foo (Foo& f);              // copy constructor
    Foo (Foo&& f)              // move constructor

    Foo& operator=(Foo& f);    // copy assignment
    Foo& operator=(Foo&& f);   // move assignment

    ~Foo();                    // destructor
};
```

in example above if we define at least one of em then compiler aint gon generate any of them for us. cus it aint secure.
if u still want compiler to make em for you, u need to use ' = default

```
class Foo {

public:
    Foo();                     = default
    Foo (Foo& f);              = default
    Foo (Foo&& f)              = default

    Foo& operator=(Foo& f);    = default
    Foo& operator=(Foo&& f);   = default

    ~Foo();                    = default
};
```

delete We can “delete” a function; that is, we can state that a function does not exist so that it is an error to try to use it (implicitly or explicitly).

`example`

```
class Foo {

    Foo (Foo& f) = delete;      // No copy constructor
};

f() {

    Foo f;
    Foo f1 {f}  // Error: no copy constructor
}


// we can also use it for functions, for e.g:

class F {

public:
    void goo() = delete;
    void foo();
};

f() {

    F f;
    f.goo();    // Error: f.goo() deleted
    f.foo();    // Ok.
}
```

well u might be wondering whats the fucking use of this shit well, lemme halla @ u real quick and tell u that one use case would be to control where the object will be stored like on the `free store` or `stack`.

`example`

```
class Not_on_free_store {

    void* operator new (size_t) = delete
};

class Not_on_stack {

    ˜Not_on_stack() = delete;
};

void f() {

    Not_on_stack v1;                                     // error : can't destroy
    Not_on_free_store v2;                                // OK.

    Not_on_stack* p1 = new Not_on_stack;                 // OK.
    Not_on_free_store* p2 = new Not_on_free_store;       // error : can't allocate
}

// However, we can never delete that Not_on_stack object. The alternative technique of making the destructor private can address that problem.
```

`virtual`

is a keyword thats specifies that the function is to be overriden by its derived class. There are two types of virtual functions *virtual* and *pure virtual*.
IMP note bout virtual functions is that if u have at least 1 virtual function u need virtual distructor for that.

`example`

```
class B {
public:
    virtual void f() { cout << " in base \n"; } // vitual
    virtual void g() = 0;                        // pure virtual

    virtual ~B();
};

void B:g() { cout << "in B \n"; }   // Error: pure virtula function

class D : B {
public:
    void f() { cout << " in D \n"; }
    void g() { cout << " in D \n"; }
};
```

`override` and `final`

override and final are whats called a contextual keyword. which means for example if u declare an object or a variable or anything with name **override** or **final** they dont mean nothin compiler is gonna treat em as just name, so to make use of em (what they meant to be used for) u need to declare em at the end of the fucntion that u wanna override. like literally at the end of the function declaration for ex. `void foo() override final {}`.

override

so override is a keyword which specifies that the function is gon get overriden. like normally people dont say override when they are overriding a virtual fucntion from the base class. but it looks cool and helps maintian the code.

```
example
```

```
class B {
    public:
        void foo();
        int override;
};

class D : public B {
    public:
        void foo() override { /* mess w/ foo */ }

        override = 12;  // again this override doesnt mean nothin its just name of an int we declared in out base class above.
};
```

final

so, in my opinion final is pretty cool cus it doesnt let u override an object again cus its litteraly final. u feel me? so the idea here is if u got a virtual function in yo base class and u wanna override it but dont want any other classes to fuck w/ it again u use override (exmaple below might explain it better ig.) and if u try to override it BOOM its an error.

```
class B {
    public:
        vitrual void f();
};

class D : public B {

    public:
        void f() override final { /* mess w/ f() */ }
}

class H : D {

    public:
        void f() override final { /* tryna mess w/ f() */ }     // BOOM its an error. can't override f() cus its declared final above
in D.
}

// u can make every virtual of a class final, like following:
class F {

    public:
        virtual void print();
}

class G final : public F {

    // ok. cool ima make every virtual function in this class final - compiler said!
    public:
        void print() override {}    // ok. cool override it.
}

class H : public G {

    void print() override {}        // error: hell nah its final, can't do it!
}
```

Smart Pointers

smart pointers are used to make sure the object is deleted if it is no longer used (refrenced)(e.g: when object goes out of scope) there are basically 2 types of STL smart pointers available for us.

unique_ptr

this template holds a pointer to an object and deleted this object when *unique_ptr<>* is deleted. so your lazy ass dont need to explicitly say *delete*, the *unique_ptr* destructor is always called so the element that u stored on the free stored is always deleted s **No Memory Leaks**.

As the name implies make sure only exactly one copy of an object exists. *unique_ptr<>* does not support copying if u try to copy u gon get compile time errors but it supports move semantcis obviously **Note: if a unique pointer already holds pointer to an existing object then that object gets deleted and new pointer is stored e.g: `unique_ptr<int> ptr; ptr.reset( new int{13} );`

The interface that unique_ptr provides is very similar to the ordinary pointer but no pointer arithmetic is allowed. tho this template calss has some helper functions w/ it they dope now u now i'm lazy af. so i aint write em all down here!

unique_ptr provides a function called release which yields the ownership. The difference between *release()* and *reset( )*, is release just yields the ownership and does not destroy the resource whereas reset destroys the resource.

```cpp
// Move Example W/ unique_ptr

#include <iostream>
#include <memory>
#include <utility>

int main() {

    std::unique_ptr<int> ptr( new int{12});
    std::unique_ptr<int> ptr2(std::move(ptr));


    std::cout << "Does ptr hold an object? "
                << std::boolalpha << static_cast<bool>(ptr) << '\n'
                << *ptr2 << '\n';

}
```

shared_ptr

The shared pointer is a reference counting smart pointer that can be used to store and pass the reference beyond the scope of a function,this is pirtucularly useful int he context of the OOP. to store a pointer as a member variable and return it to access the value outside the scope of the class.

The usage of shared pointer easily pass and return refrence to objects w/t running into memory leaks or invalid attempts to access deleted references "they r thus a cornerstone of modern memory management" - cppreference.com

u can also use std::make_shared<T>() to assign the values and ptr.use_count() to get the reference count.

```cpp
#include <iostream>
#include <memory>

class Foo {
    public:
        void dosmn() { std::cout << "SMN()\n"; };
};

class Bar {

    public:
        Bar() {
            pfoo = std::shared_ptr<Foo>(new Foo());
        }

        // return shared pointer to Foo object
        std::shared_ptr<Foo> getFoo() { return pfoo; }

    private:
        std::shared_ptr<Foo> pfoo;
};

void doSmn() {

    Bar* pBar = new Bar();    // w/ the Bar object new Foo is created and stored
    // reference count = 1

    std::shared_ptr<Foo> pFoo = pBar->getFoo(); // a copy of shared pointer is created
    // reference count = 2

    pFoo->dosmn();

    delete pBar; // with pBar the private pFoo is destroyed
    // reference count = 1

    return; // w/ the return the local pFoo is detsriyed automatically
    // reference count = 0

    // Internally the std::shared_ptr detsroyes the refernce to the Foo object
}

void doSmnElse(std::shared_ptr<Bar> pBar) {

    std::shared_ptr<Foo> pFoo = pBar->getFoo(); // a copy of shared pointer is created
    // reference count = 1

    pFoo->dosmn();

    return; // local pFoo is destroyed
}

int main() {

    doSmn();

    std::shared_ptr<Bar> pBar (new Bar);
    doSmnElse(pBar);

}
```

by default shared_ptr called delete even if u have an array allocated, so to overcome this problem we can use a lambda.
`std::shared_ptr<int> arr (new int[3], [](int* p) { delete[] p; } );`

weak_ptr

A weak pointer provides sharing semantics and not owning semantics. This means a weak pointer can share a resource held by a shared_ptr. So to create a weak pointer, some body should already own the resource which is nothing but a shared pointer.

A weak pointer does not allow normal interfaces supported by a pointer, like calling *, ->. Because it is not the owner of the resource and hence it does not give any chance for the programmer to mishandle it. Then how do we make use of a weak pointer?

The answer is to create a shared_ptr out of a weak_ptr and use it. Because this makes sure that the resource won't be destroyed while using by incrementing the strong reference count. As the reference count is incremented, it is sure that the count will be at least 1 till you complete using the shared_ptr created out of the weak_ptr. Otherwise what may happen is while using the weak_ptr, the resource held by the shared_ptr goes out of scope and the memory is released which creates chaos.

```
void main( )
{
    shared_ptr<Test> sptr( new Test );  // ref count = 1
    weak_ptr<Test> wptr( sptr );        // ref count = 1, weak count = 1
    weak_ptr<Test> wptr1 = wptr;        // ref count = 1, weak count = 2

    // Assigning a weak pointer to another weak pointer increases the weak reference count.

    // to get a shred pointer from weak pointer we could use lock().
    shared_ptr<Test> sptr( new Test );  // ref count = 1
    weak_ptr<Test> wptr( sptr );        // ref count = 1, weak count = 1
    weak_ptr<Test> wptr1 = wptr.lock();        // ref count = 1, weak count = 1
}
```

alt text

---

boost::asio

some quick notes bout bost asio, so i hade a basic asio progarm and it took me 100 hours to figure out which files i need to link in order to compile (found stack overflow post - thanks to the answerer) and these are the three files u need to link in order to use boost::asio.

`c++ -I /path/to/boost_1_67_0 example.cpp -o example -lpthread -lboost_system -lboost_signals ( -lboost_thread - if u are using thread.)`

please refer to [boost (boost.org/doc/boost_asio/tutoria)](boost.org/doc/boost_asio/tutoria) for greaer details

---

Pointer to members

pointers-to-member operators, `.*` and `->*` **return the value of a specifc calss member.** for the objec specified on the left side of the expression. the right side must specify a member of the calss.

`example`

```
#include <iostream>

class F {
public:
    void foo() {
        sd::cout << "foo\n";
    }

    int num;
};

void (F::*p2mf)() = &F::foo;    // pm2f is a pointer to member foo()
int F::*p2mv = &F::num;         // p22mv is a pointer to member num

main() {

    F f;            // create object of type F
    F* fp = new F;  // pointer to object of type F

    (f.*p2mf)()     // Invoke foo() from f
    (fp->*p2mf)()   // Invoke foo() from fp

    f.*p2mv = 13;   // f's  num -> 13
    f->*p2mv = 16;  // fp's num -> 16

    std::cout << "f's num: " << f.*p2mv
              << "fp's num: " << fp->*p2mv
              << '\n';
}
```

`example w/ using`

```
class B {
public:
    virtual void foo() {
        std::cout << "From Base\n";
    }
};

class D : public B {
public:
    void foo() override {
        std::cout << "From D\n";
    }
};

using bptr = void(B::*)();

int main() {

    /*
        B* b = new B;

        bptr bp = &B::foo;
        (b->*bp)(); // invoke B's foo()

    */

    B* b = new D;

    bptr bp = &B::foo;
    (b->*bp)(); // invoke D's foo()

    delete b;
}
```

> just follow the same rule of '->' (for pointers) and '.' (for non pointers) but dont forget the '*'.

pointer to member is not like a "normal pointer" its different it doesnt point to an address. like normal variable and function pointers do. instead it is more like an offset into a structure or an index into an array.

> A static member isn't associated with a particular object, so a pointer to an ordinary pointer. For example: your porn collection

---

Inheritance

Access Control

- If it is **private** , its name can be used only by member functions and friends of the class in which it is declared.
- If it is **protected** , its name can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class
- If it is **public** , its name can be used by any function.

Modes of inheritance

- `public mode`: if we derive a public class from a base class then the **public** members of the base class will become **public** in derived class and **protected** will become **protected**.
- `protected mode`: if we derive sub class from a **proteced** base class then both **public** member and **proected** members of the base class will become **protected** in derived class.
- `private mode`: if we derive sub class from a **private** base class then both **public** member and **protected** members of the base class will become **private** in base class.

```
class B {

    protected:
        int y;

    public:
        int z;

    private:
        int x;

};

class D1 : public B {
    // z is public
    // y is protected
    // x is inaccessible
};

class D2 : protected B {
    // z is protected
    // y is protected
    // x is inaccessible
};

class D3 : protected B {
    // z is private
    // y is private
    // x is inaccessible
};
```

> use proteced only when u will die if u dont use it. and dont declare members protected.

Virtual base classes

virtual base classes are used in virtual inheritance as a way of preventing multiple **instances** of a given class in an inheritance hierarchy when using multiple inheritances.

few details about em:

- virtual base classes are always created before non-virtual base classes, which insures that all bases get created before their derived classes
- if a class inherits one or more classes that have virtual parents, the *most* derived class is responsible for constructing the virtual base class.

`example w/t virual base`

```
class A { };

class B : public A { };

class C : public A { };

class D : public B, public C { };

// our class hierarchy is:

     _________                               _________
    | class A |                             | class A |
    |_________|                             |_________|

        ^                                       ^
        |                                       |
        |                                       |
        |                                       |
        |                                       |
     _________                               _________
    | class B |                             | class C |
    |_________|                             |_________|
       ^                                       ^
       |                                       |
       |_______________        ________________|
                       |      |
                      _________
                     | class D |
                     |_________|
```

`example w/ virual base`

```
#include <iostream>

class Storable {
  public:
    Storable(const std::string& s); // store in file named s
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~Storable();

  protected:
    std::string file_name;
    Storable(const Storable&) = delete;
    Storable& operator=(const Storable&) = delete;
};

class Transmitter : public virtual Storable {
  public:
    void write() override;
    // ...
};

class Receiver : public virtual Storable {
  public:
    void write() override;
    // ...
};

class Radio : public Transmitter, public Receiver {
  public:
    void write() override;
    // ...
};


// now our hierarchy is:


                                     ________________
                                    | class Storable |
                ________________    |________________|   __________________
               |                                                           |
               |                                                           |
             ________________                                        ___________________
            | class Reciever |                                      | class Transmitter |
            |________________|                                      |___________________|
                        ^                                                ^
                        |                                                |
                        |________________        ________________|
                                        |      |
                                     _____________
                                    | class Radio |
                                    |_____________|

/*
Note that if we override one virtual function in one base class then we need to do it for all of em or its gon give us the error
saying:
"override of virtual funcion "Storabel::write is ambigious"
*/
```

one more example (cus it aint like i got a job or smn)

```
#include <iostream>

struct V {
  V(int i);
  // ...
};

struct A {
  A(); // default constructor
  // ...
};

struct B : virtual V, virtual A {
  B() :V{1} { /* ... */ }; // default constructor ; must initialize base V
  // ...
};

class C : virtual V {
  public:
    C(int i) : V{i} { /* ... */ }; // must initialize base V
    // ...
};

class D : virtual public B, virtual public C {

    // implicitly gets the virtual base V from B and C
    // implicitly gets virtual base A from B

  public:
    D() { /* ... */ } // error : no default constructor for C or V
    D(int i) :C{i} { /* ... */ }; // error : no default constructor for V
    D(int i, int j) :V{i}, C{j} { /* ... */ } // OK
    // ...
};

// well i think now yall can guess the hierarchy of these if not then fuck bruh u dumb as hell.


     _________                                _________
    | class V |                              | class A |
    |_________|                              |_________|

        ^                                         ^
        |    _____________________________________|
        |___|_____________________________________
        |   |                                     |
        |   |                                     |
     _________                                _________
    | class B |                              | class C |
    |_________|                              |_________|
        ^                                        ^
        |                                        |
        |_______________         ________________|
                        |       |
                      _________
                     | class D |
                     |_________|

// Fuck it took me shit ton of time to make this shit.
```

Tip: Use a virtual base to represent something common to some, but not all, classes in a hierarchy.

Runtime Type Information (RTTI)

the use of type information at runtime is reffered to as RTTI. there are 3 main Cxx language elements to runtime type information

- dynamic_cast used for conversion of polymorphic types
- typeid used for identifying exact type of an object
- type_info used to hold the type information returned by the typid operator

example

```
void my_event_handler(BBwindow* pw)
{
    if (auto pb = dynamic_cast<Ival_box*>(pw)) { // does pw point to an Ival_box?
        // ...
        int x = pb->get_value(); // use the Ival_box
        // ...
    }
    else {
        // ... oops! cope with unexpected event ...
    }
}
```

Casting from a base class to a derived class is often called a downcast because of the convention of drawing inheritance trees growing from the root down. Similarly, a cast from a derived class to a base is called an upcast. A cast that goes from a base to a sibling class, like the cast from BBwindow to Ival_box, is called a crosscast.

typeid

syntax

```
typeid(type-id)

typeid(expression)
```

the **typeid** operator allows type of the object to be determined at runtime. The expression must point to a polymorphic type (w/ virtual functions) otherwise the result is the type_info for the static class reffered to int the expression. and pointer must be derefferenced so that object it points to is used w/t derefferencing the result will be the type_info for the pointer not what it points to

example

```
#include<typeinfo>

class B {
public:

    virtual void foo() {}
};

class D : public B {

public:

    void foo() {

    }
};

int main() {

    D* d  = new D;
    B* b  = d;

    std::cout << typeid(d).name() << '\n';
    std::cout << typeid(*d).name() << '\n';
    std::cout << typeid(b).name() << '\n';
    std::cout << typeid(*b).name() << '\n';

    // name() returns a C Style sring which resides in memory owned by the system so dont tryna act smart and 'delete[]' it.
}

// Output:
class D *
class D
class B *
class D
```

if the expression is derefferencing a pointer and pointer's value is 0, typeid throws **bad_typeid** exception and if the pointer does not point to the valid object **__non_rtti_object__** exception is thrown.

if expression is not a pointer niether a reference to a base class of he object the result is static type of the expression and note that *static type referes to the type of an expression known at compile time* reference (somtimes) and execution semantics are ignored when evaluating static type.

example

```
#include <typeinfo>

main() {

    std::cout << ( typeid(int) == typeid(int&) ? "True" : "False") << '\n';

    // no wonder this bitch is true.
}
```

the cool thing is we can know the types of them expressions, its so fucking helpfull when using w/ templates

```cpp
#include<iostream>
#include<typeinfo>
#include<string>

using namespace std::string_literals;

template<typename T>

void checkType(T t) {


    if (typeid(t).name() == "int"s)
        std::cout << "Type: int\n";

    else if (typeid(t).name() == "const char *"s)
        std::cout << "Type: C-Style String" << '\n';

    else if (typeid(t).name() == "class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >"s)
        std::cout << "Type: Cxx String" << '\n';
}

int main() {


    checkType("smn"s);  // Cxx String
    checkType(12);      // int
    checkType("smn");   // C-Style String

    return 0;
}

/* Fucking Cool Eh? */
```

type_info::name returns a human readable format of the type and **type_info::raw_name** returns ugly but its litstle faster than the other one so when u dont a give a shit bout performance use **name** and when u do use **raw_name**. ps: this is what **type_info::raw_string** of a std::string looks like ".?AV?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@" Welcome To Cxx People!

Search for files in a specific directory

Non Recursively

don't trip on it non-recursively just means it returns the files from the folder that u want. for example i got a file structure like his

```
/ Folder
    / file1
    / folder
         / file1
        / file2
    / file2
```

so now its gonna return file1 and file2 and "folder" u feel me!

`example:`

```cpp
#include <iostream>
// if your compiler has support just include <filesystem>
#include<experimental/filesystem>
#include <string>

namespace fs = std::experimental::filesystem;


int main() {

    std::string path = "/path";

    for (const auto& entry : fs::directory_iterator(path))
        rt << entry.path() << '\n';

    return 0;
}
```

Recursively

man this shit is lit. recursive iterator returns filesnames/foldernames from folder we specified and also from the subfolders for example lets assume we got this dir heirarchy

```
/ Folder
    / file1
    / folder
        / file1
        / file2
    / file2
```

so now its gonna return file1, file2, folder, folder/file1, folder/file2 u feel me!

```
#include <iostream>
// if your compiler has support just include <filesystem>
#include <experimental/filesystem>
#include <string>

namespace fs = std::experimental::filesystem;


int main() {

    std::string path = "/path";

    for (const auto& entry : fs::recursive_directory_iterator(path))
        rt << entry.path() << '\n';

    return 0;
}
```

TIP: dont forget to link fs by **-lstdc++fs** if you are using gcc or clang.

Templates

templates are dopeass way to kinda do generic programming in C++ or its a way to work w/ generic types. it simply allows us to not repeat the same code for different data types.

"If the algorithm is same but data types are different, there aint no need to overload function instead its time to chill yo ass down and use templates" - Bjarne Strostrup and me. :p

Function Templates

`syntax`

`template<class identifier> function-decleration`

`example`

```
template<class T>
T sum(T t) {
    return t + t;
}

int main() {

    return sum(12);
}
```

in the above example we didnt need to say sum<int>(12) cus it got deduced automaically thanks to our litass compilers but what if 12 was a real number then we got two options:

1. `sum<double>(12);`
2. `sum(12.0);`

either of em works.

when the compiler ecounters than we are using template in this case a "function template" it replaces every occurence of T w/ the type than we passed to it. did ya get that only if we using em i mean thats fucking awesome and everything is done at compile time so 0 runtime overhead ps: i still donno what the fuck that means but it sounds cool so im down for it.

Class Templates

dont shit yoself after hearing this shit cus it just means a class that has template memebers. could be a variable a fucntions etc.

`example`

```cpp
#include <iostream>

template<class T>
class Foo {
    T val;

    public:
        Foo() : val{} { }
        Foo(T _val) : val{_val} { }
        T getVal();
};

template<class T>
T Foo<T>::getVal() {
    return val;
}

int main() {

    Foo<int> foo{12};
    Foo<char*> f{"Thoties be crazy!"};
    std::cout << f.getVal() << ' ' foo.getVal();

    return 0;
}

// output
Thoties be crazy! 12
```

if u are defining your member function that is "templated" man idk the right word for it but u got the point right? so in that case u gotta define your template again. cus scope of out first template is limited till closing curly bracket of our class.

class template specialization

when we want to define a different implementation for a template when a specific type is passed we can declare a specialization for that. don't worry u'll get the hang of it when u see the example below.

`example`

```cpp

/* Assume
 * Above Foo is Class Here.
 * we could just make another version of it to support  concatination of char*(s)
 */

template<>
class Foo<char*> {

  public:
    Foo() : val{} { }
    Foo(char* _val) : val{_val} {}
    void concat(const char*);
    char* getVal() { return val; }

    ~Foo() {
      delete val;
    };

  private:
    char* val;
};


// **Disclaimer** aint the right way to concat ig.
void Foo<char*>::concat(const char* str) {

  int beg = strlen(val);
  int end = strlen(str) + beg;

  char tmp[end];

  for (int i = 0; i != beg; ++i)
    tmp[i] = val[i];

  for (int i = 0, j = beg; j != end - 1; ++i, ++j)
    tmp[j] = str[i];

  val = tmp;
}

int main() {

  Foo<char*> f("Thoties be crazy");
  f.concat(" 4 Real.");

}
```

Value Parameters

we could also have our good old normal ass data types in templates and thats what this shit is about.

example

```
#include <iostream>

template<class T, std::size_t N>
void foo(T(&)[N]) {
  std::cout << "Size of array: " << N;
}

int main() {

  int x[] = {1, 2, 3, 4, 5, 2, 1, 2};
  foo(x);

  return {};
}

// man this shit is fucking cool!
```

anoher example

```
#include <iostream>
#include <stdexcept>

template<class T, std::size_t N>
class Bar {

  public:

    Bar(std::initializer_list<T> val) {

      int i = 0;
      for (auto x : val) {
        arr[i] = x;
        ++i;
      }

    }

    void print() {
      for (int i = 0; i != N; ++i)
        std::cout << arr[i] << '\n';
    }

    void insert(std::uint64_t i, T val) {
       if (i >= N)
         throw std::out_of_range{"cannot insert, index out of bound!"};

      arr[i] = val;
    }

  private:
    T arr[N];
};

int main() {

  Bar<int, 5> a{1, 2 ,3, 4, 5};
  a.insert(6, 12);
  a.print();

  return {};
}
```

Default Values

yes u can set default values to templates. neat eh?

example

```cpp
#include <iostream>
#include <stdexcept>

template<class T = double, std::size_t N = 12>
class Bar {

  public:

    Bar() : arr{0} { } // set every elem to 0
    Bar(std::initializer_list<T> val) {

      int i = 0;
      for (auto x : val) {
        arr[i] = x;
        ++i;
      }

    }

    void print() {
      for (int i = 0; i != N; ++i)
        std::cout << arr[i] << '\n';
    }

    void insert(std::uint64_t i, T val) {
       if (i >= N)
         throw std::out_of_range{"cannot insert, index out of bound!"};

      arr[i] = val;
    }

  private:
    T arr[N];
};

int main() {

  Bar<> bar; // same as Bar<double, 12>
  bar.insert(0, 120);
  bar.print();

  return {};
}
```

all we did in above example was add default values our types. and it works like charm.

Operations as arguements

we could use functions pointers or smn in the templates and smn more i guess cus i aint Cxx God. tho using function pointers is fucking cool.

`example`

```cpp
#include <iostream>

template<class T, bool(*validate)(T t, T t2)>
void foo(T t1, T t2) {

  std::cout << std::boolalpha << validate(t1, t2) << '\n';
}

int main() {

  auto val = [](int a, int b) {
    return a > b;
  };

  foo<int, val>(12, 113);

  return 0;
}
```

simple but demons what im tryna say i guess. we could make this shit more complex like sorting in std::map yes it uses shit like this too. for example we could define our own implemenation for sorting of keys in std::map its perticularly usefull when we are dealing the case where some user defined type is the key for yo map or smn.

`example w/ std::map`

```
#include <iostream>
#include <map>
#include <string>

using std::string;

struct Person {

    Person() = default;
    Person(string name, int age) : _name{ name }, _age{ age } { }

    string getName() const { return _name;  }

    private:
        string _name;
        int _age;
};

int main() {


    auto sort = [](auto a, auto b) {
        return a.getName() < b.getName();
    };

    std::map<Person, int, decltype(sort)> map{sort};

    map[Person("Kanye", 12)]    = 12;
    map[Person("T Pain", 33)]   = 33;
    map[Person("Biggy", 8)]     = 8;
    map[Person("Remy Ma", 21)]  = 21;


    for (auto& x : map) {
    std::cout << "Values: " << x.second << '\n';
    }

    system("pause");

    return 0;
}
```

default sorting for keys is "less than" i.e Ascending Order if all we want to keep em in Descending order its pretty easy to do

`example`

```
std::map<std::string, int> map{};                           // Default std::less
// or std::map<std::string, int, std::less<std::string>> map{};

std::map<std::string, int, std::greate<std::string>> map{}; // From greater to lower

std::map<double, int, std::greater<double>> map{};          // w/ double's
```

MetaProgramming

programming that manipulates entities such as classes and functions is commonly called as metaprogramming. yeah if you didnt got nothin from that word dont worry i didnt too. just think of templates as classes and function generators and metaprogramming as a way to do computation at compile time.
ps: Metaprogramming is also called as " Generative Programming ", " Two Level Programming ", " Multilevel Programming " etc.

Two main reasons for using metaprogramming

- Improved type safety - we can get the exact type for a data structure so we dont need to do casts and shit.
- Improved runtime performance - we can do computation at compile time and select functions to be called at compile time that way we can eliminate runtime overhead. pretty cool!

Type Functions

a type function is a function that takes at least one type arg or produces at least one type as a result.

```
#include <iostream>
#include <type_traits>
#include <typeinfo>

int  main() {

    enum class Axis : char { x, y, z };
    enum flags { off, x = 1, y = x << 1, z = x << 2, t = x << 3 };

    typename std::underlying_type<Axis>::type axisType;
    using flagType = typename std::underlying_type<flags>::type;

    std::cout << typeid(axisType).name();   // char
    std::cout << typeid(flagType).name();   // int
}
```

this shit gets very handy when used w/ smn like std::conditional or std::enable_if

well for sake of my keyboard lets make our won type function

```
#include <iostream>
#include <type_traits>
#include <typeinfo>
#include <string>

template <unsigned N, typename... cases> struct Select;

template <unsigned N, typename T, typename... cases>
struct Select<N, T, cases...> : Select<N - 1, cases...> {

};

template <typename T, typename... cases>
struct Select<0, T, cases...> {
    using type = T;
};


template<unsigned N, typename... Cases>
using select = typename Select<N, Cases...>::type;


int main() {

    using Type = select<4, int, char, double, char*, std::string>;

    Type sayIt = "Hey i am std::string";

    std::cout
        << typeid(Type).name()  << '\n'
        << sayIt                << '\n';
}
```

the above example works kinda like std::conditional but it could choose between N number of types. but the imp thing this example illustrates is recursion and the dopeass variadic templates. btw above example is copied from bjare's Cpp programming language book did u really think that was my shit? thank yo but no lmao.

Type Predicates

A predicate is a function that returns a Boolean value

`example`

```
#include <type_traits>

template <typename T>
void foo(T t) {

    if (std::is_polymorphic<t>::value) {
        // do something if its true
    } else {
        // do something if its false
    }

    // the ::value in is_polymorphic is a boolean value.
    // // and i aint got write the workings and shit of is_polymorphic here cus i wrote smn bout it in the standard library section
and also cus im lazy.
}
```

if u dont wanna do ::value theres always _v version of these type functions

```
#include <iostream>
#include <type_traits>

struct Foo {
    virtual void foo() { }
};

struct Bar {
    void bar() { }
};

int main() {

    std::cout
        << std::boolalpha
        << std::is_polymorphic_v<Foo>   // true
        << std::is_polymorphic_v<Bar>   // false
        << '\n';
}
```

so now lets make a type predicate function to check if the given type is c style string or not

```
#include <iostream>
#include <type_traits>
#include <string>

template <typename T>
struct is_c_style_string : std::false_type { };

template <>
struct is_c_style_string<char*> : std::true_type { };

template <>
struct is_c_style_string<const char*> : std::true_type { };


template <typename T>
bool is_c_string(T t) {

    return is_c_style_string<T>::value;
}


int  main() {

    using namespace std::literals::string_literals;

    std::cout
    << std::boolalpha
    << is_c_string("Hello const char*")      // true
    << '\n'
    << is_c_string("Hello std string."s)     // false
    << '\n'
    << is_c_string(12)                       // aphcourse false
    << '\n';

}
```

in the example above std::false_type and std::true_type are just two structs defined in Cxx standard library and them structs have ::value and ::type defined in them so basically if we inherit from em false_type inherited's value is false and true_type's value true. its quite handy, but as u can see in main() we cannot directly pass a string as a template parameter to our is_c_style_string<> thats why i defined is_c_string() helper function which takes in a type T and returns the ::value inside is_c_style_string<> depending on the specialization.

Recursion

recursion is kidna the big part of metaprogramming cus this is how we can loop at compile time. so like always lemme take an classic example of fibonacci sequence.

```
#include <iostream>
#include <type_traits>
#include <typeinfo>

template <int N>
struct Fib {
    const static int value = Fib<N - 1>::value + Fib<N - 2>::value;
};

template <>
struct Fib<2> { const static int value = 2; };

template <>
struct Fib<1> { const static int value = 1; };

template <>
struct Fib<0> { const static int value = 1; };

int main() {

   return Fib<10>::value;   // 8
}
```

the example above looks pretty hacky and cool but plz dont do it in real life code its cool to just mess around w/ it. this isnt the code u would wanna write 4real. u should use constexpr functions instead of this shit then. tho lets get to this example so when we are saying value = Fib<N - 1>::value + Fib<N - 2>::value we are in simple word using recursion.

lets say if wanted to find out 5th fibonacci number (hope i spelled it right).

1. Fib<5> - value = Fib<4> + Fib<3> -> this means it needs Fib of 4 and 3 so its gon create em
2. Fib<4> - value = Fib<3> + Fib<2> -> in needs Fib of 3 and 2 so create 3 but dont create 2 cus we got specialization for that
3. Fib<3> - value = Fib<2> + Fib<1> -> dont need to create em cus we got specializations for values <= 2

Now that we have everyting we need we go from botton to top w/ the values

3. value = Fib<2>::value + Fib<1>::value -> 2 + 1
4. value = Fib<3>::value + Fib<2>::value -> 3 + 2
5. value = Fib<4>::value + Fib<3>::value -> 5 + 3

yeah i could say templates bring shit ton of complexity to the table

lemme put another example just for the sake of complexity

```cpp
#include <iostream>

template <int N>
constexpr int Fact() {
    return N * Fact<N - 1>();
}

template <>
constexpr int Fact<1>() {
    return 1;
}

int main() {

    return Fact<5>();   // 120
}
```

the code above is a fucking nightmare dont do this shit instead u can do smn like this

```cpp
constexpr int fac(int i) {

    return (i < 2) ? 1 : fac(i - 1);
}

constexpr int f3 = fac(3);

// #beautifull
```

enable_if

std::enable_if is an awsome way to provide a perticular functionality for templates kinda like the specialization

The good exmaple to demonstrate the use case for enable_if would be lets say u got a smart pointer class like following:

```cpp
template<typename T>
class Smart_pointer {
    // ...

    T& operator*();      // return reference to whole object
    T* operator->();     // select a member (for classes only)

    // ...
}
```

If T is a class, we should provide operator−>(), but if T is a built-in type, we simply cannot do so (with the usual semantics). Therefore, we want a language mechanism for saying, “If this type has this property, define the following.” We might try the obvious

```cpp
template<typename T>
class Smart_pointer {
    // ...

    T& operator*(); // return reference to whole object
    if (Is_class<T>()) T* operator->(); // syntax error
    // ...
}
```

but this shit doesnt work C++ does not provide an if that can select among definitions based on a general condition and this is where enable_if comes into play.

```cpp
template<typename T>
class Smart_pointer {
    // ...

    T& operator*(); // return reference to whole object

    template <typename U = T>
    std::enable_if_t<std::is_class_v<U>, U>* operator-> () { return t; }
    // ...
}
```

here if type U whihc is indeed T is a class then std::enable_if is gon define operator -> else no. i had o define another template inside Smart_pointer class and assing type U to be T cus if we dont do that we gon get the compile time error cus the case where T is a built in type enable_if isnt gon have type defined in it.

Another and prolly the more common way to use enable_if is in the templte<>.

```
#include <iostream>
#include <type_traits>

template <typename T, typename = std::enable_if_t<std::is_integral_v<T>, T>>
T foo(T t) {
    return t * t;
}

int main() {

    auto i = foo(12001l);   // OK. its type is integral
    auto ii = foo(12);      // OK. aphcourse
    auto c = foo('c');      // OK. works for chars cus they are integral

    double d = foo(12.9);   // Error. arguements does not match

    const char* s = foo("Ye for president");    // Error. arguements does not match
}
```

quite dumb example but it shows the point i guess.

Variadic Templates

a template with at least one parameter pack is called as variadic template well aphocourse now ya wanna know whats a parameter pack welll lemme copy this shit from cppreference.com " A template parameter pack is a template parameter that accepts zero or more arguements (non-types, types, templates).
A function parameter pack is a parameter that accepts zero or more function arguements.

```
#include <iostream>

template <typename T>
void foo(T t) {
    std::cout << t;
}

template <typename T, typename... Args>
void foo(T val, Args... args) {
    std::cout << val << '\n';
    foo(args...);
}


int main() {

    foo("Hello World", 12, "E is best rapper of all time!", 12.99, 'c');

    // output
    /*
        Hello World
        12
        E is best rapper of all time!
        12.99
        c
    */
}
```

so, lemme tryn break this shit down what i think is happening is first we are printing the first value in the pack and then when we call `foo(args...)` its extracting the pack and now arguements will be `foo(12, ...)` and when the only one value is left `foo(T t)` is printing it.
if we tryn imagine this shit in the from of steps this'll look kinda like this ig.

1. `foo("Hello Word", rest of the args...)`

   print 1st arguement i.e "Hello World"

2. `foo(12, rest of the args...)`

   print 1st arguement i.e 12 now

3. `foo("E is best rapper of all time!", rest of the args...)`

   print 1st arguement i.e "E is best rapper of all time!" now

4. `foo(12.99, rest of the args...)`

   print 1st arguement i.e 12.99 now

5. `foo('c')`

   see there are no arguements other than c now so compiler isn't gon call `foo(T t, Args... args)` its gon call `foo(T t)`
   see not the difficult!

key to understanding variadic template is to undersand pack expansion when we call `foo(args...)` recursively in our `function ...` after args means yo i want this pack to be expanded so compiler will do the work for u and send arguements as `foo(1st arguement, rest of the pack...)` to the function.

tbh im too lazy write down all the rules and shit u should go here and get the complete reference.

Forwarding

std::forward returns an rvalue ref to the arg if the arg is not an lval ref and if an arg is lvalue it retuns arg W/t modifying it. and One of the major uses of variadic templates is forwarding from one function to another

```
#include <iostream>

template <typename T, typename... Args>
void call(T&& t, Args&&... args) {

    t(std::forward<Args>(args)...);
}

void foo(int a, int b) {

    std::cout << a << ' ' << b << '\n';
}

int main() {

    call(foo<int>, 12, 12);
}
```

our call functioint is kida intike std::bind where u give it the function and then give the arguements for that function